AD-A260 713

(2)

NASA Contractor Report 189729
ICASE Report No. 92–60

DTIC
ELECTE
JAN 2 6 1993
S C D

ICASE

PARALLEL ALGORITHMS FOR SIMULATING CONTINUOUS TIME MARKOV CHAINS

David M. Nicol
Philip Heidelberger

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

NASA Contract Nos. NAS1–18605 and NAS1–19480
November 1992

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23681–0001

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

93-01345

24pf

93 7 07

DTIC QUALITY INSPECTED 5

Parallel Algorithms for Simulating Continuous Time Markov Chains

David M. Nicol *
Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23185

Philip Heidelberger
IBM Thomas J. Watson Research Center, Hawthorne
P.O. Box 704
Yorktown Heights, New York 10598

Accession For
NTIS [illegible] ☑
[illegible] ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Abstract

We have previously shown that the mathematical technique of uniformization can serve as the basis of synchronization for the parallel simulation of continuous-time Markov chains. This paper reviews the basic method and compares five different methods based on uniformization, evaluating their strengths and weaknesses as a function of problem characteristics. The methods vary in their use of optimism, logical aggregation, communication management, and adaptivity. Performance evaluation is conducted on the Intel Touchstone Delta multiprocessor, using up to 256 processors.

*Research was supported in part by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-18605 and NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681. This work was initiated while David Nicol was a visiting scientist at the IBM T.J. Watson Research Center. Partial support was provided by NSF Grants ASC 8819373 and CCR-9201195, and NASA Grants NAG-1-1060 and NAG-1-995.

1 Introduction

Discrete-event simulation is an invaluable tool for the design and analysis of complex systems such as factories, transportation networks, computer systems, and communication networks. Large scale simulations require a long time to execute, and because of this many researchers are interested in parallelizing their execution. One of the key issues is synchronization between processors, as the synchronization demands are highly variable, depending dynamically on the simulation model's state. Recommended introductory surveys on the topic are found in [2] and [11].

In a series of previous papers [5, 9, 10] we developed the notion of using *uniformization* as the basis for synchronization in parallel discrete-event simulation of continuous-time Markov chains (CTMCs). CTMC models are important, appearing frequently in the study of computer and communication systems. Uniformization exploits the mathematical structure of these models, making it possible to pre-compute instants in simulation time where Logical Processes (LPs) ought to synchronize. The decision whether an LP actually influences another at one of these instants is left until run-time. Conceptually, a simulation is performed in three phases. In the first phase, the simulation model is partitioned into LPs, which are mapped to processors. All simulation activity associated with an LP is assumed to be performed by its assigned processor. In the second phase one randomly generates synchronization points; in the third phase one simulates a mathematically correct sample path through those points. We call the general method PUCS, for *Parallel Uniformized Continuous-time Simulation.*

We have developed five different variations of PUCS that differ in their treatment of LP aggregation, communication management, use of optimism, and generation of communication schedules:

- Conservative Aggregated PUCS (CA-PUCS),
- Conservative Partitioned PUCS (CP-PUCS),
- Optimistic PUCS (Opt-PUCS),
- Adaptive Conservative Aggregated PUCS (ACA-PUCS).
- Adaptive Conservative Partitioned PUCS (ACP-PUCS).

CA-PUCS uses no optimism, and treats the entire submodel assigned to a processor as a single LP. Synchronization between LPs is thus equivalent to synchronization between processors. CP-PUCS also eschews optimism, but permits a processor's submodel to be viewed as a collection of LPs that are resident on the same processor. Opt-PUCS also allows multiple LPs per processor, synchronizes optimistically, and uses techniques to reduce state-saving and perform smart on-processor scheduling; these techniques are made possible by the basis in uniformization. ACA-PUCS is like CA-PUCS, except that it attempts to reduce some overheads associated with synchronization, and requires that one know less about the simulation model. Similarly, ACP-PUCS is an adaptive version of CP-PUCS.

Each of these methods has strengths and weaknesses that are alternatively revealed by problem characteristics. The object of this paper is to give an overview of uniformization-based

synchronization, and empirically examine these different methods on the Intel Touchstone Delta multiprocessor[7], using up to 256 processors.

The remainder of the paper is organized as follows. Section 2 gives an overview of direct Markovian simulation, and uniformization. Section 3 introduces each method and its rationale. Section 4 presents and analyzes our experimental results, and Section 5 gives our conclusions.

2 Uniformization-Based Synchronization

In this section we briefly describe the basic notions of direct Markovian simulation, and uniformization. More rigorous and complete mathematical details can be found in [5]. Following the descriptions we illustrate them concretely with an example.

Let us first review some basic elements of the theory of continuous time Markov chains. Readers unfamiliar with CTMCs are encouraged to consult Ross [12] for a more complete and exact introduction to the topic. A CTMC is a stochastic process $\{X(t)\}$, where $X(t)$ is the *state* of the CTMC at time t. For the purposes of general description, $X(t)$ is taken to be a nonnegative integer; in practice it is often more natural to describe $X(t)$ as a vector of integers, e.g., the vector of queue lengths in a network. Upon entering a state $\mathbf{s}$ at time t, the CTMC remains in that state for a random period of time called the *holding time*, which has an exponential distribution with state-dependent rate $\lambda(\mathbf{s})$. This is also called the *transition rate* out of state $\mathbf{s}$. At the end of the holding time, the CTMC randomly changes state, jumping to some state $\mathbf{s}'$. It is convenient to think of this jump as choosing a winner among all possible jumps, in the following way. While in state $\mathbf{s}$ the chain is attempting to make a transition to every other state reachable from $\mathbf{s}$, simultaneously. It is as though there are a large number of stochastic processes–one for each state distinct from $\mathbf{s}$—that are all concurrently active. The transition rate for the process attempting to jump to $\mathbf{s}'$ is some $q_{\mathbf{ss}'}$; note that $\lambda(\mathbf{s}) = \sum_{\{\mathbf{s}' \neq \mathbf{s}\}} q_{\mathbf{ss}'}$. Each of these processes has an exponentially distributed holding time; the rate of $\mathbf{s}'$'s holding time is just $q_{\mathbf{ss}'}$. We may imagine that each of these holding times are randomly sampled at the point $\{X(t)\}$ enters $\mathbf{s}$. Now the time and nature of $\{X(t)\}$'s transition out of $\mathbf{s}$ are defined by the process whose next transition time is least among all possibilities. Thus, the probability that the exponential associated with a given state $\mathbf{s}'$ is least among its peers is just $p_{\mathbf{ss}'} = q_{\mathbf{ss}'}/\lambda(\mathbf{s})$; $p_{\mathbf{ss}'}$ is also known as a transition probability.

Observe that we can also interpret a transition in terms of $\{X(t)\}$ simultaneously attempting jumps to one of a number of *sets* of states. For example, we might partition the state-space into two sets A and B, and interpret transition as the competition between all transitions to states in A, and all transitions to states in B. This interpretation will be particularly useful when A is the set of transitions that do not affect other LPs, and B is the set of transition that do.

A direct simulation of a CTMC involves sampling holding times, and choosing transitions, as follows. Upon entering state $\mathbf{s}$, one advances time by sampling an exponential with rate $\lambda(\mathbf{s})$, essentially simulating the duration of time the CTMC remains in state $\mathbf{s}$. To choose a transition it is not necessary to choose the least of a large number of exponentials. It suffices to construct the transition distribution by computing the rates $q_{\mathbf{ss}'}$, and then sampling from the distribution

$\{q_{ss'}/\lambda(s)\}$.

Uniformization of a CTMC is a mathematical device (originally used to simplify numerical solution [4]) designed so that every holding time is drawn from the same distribution. The basic idea is to find a *uniformization rate* λ_{max} such that for every state s, $\lambda(s) \leq \lambda_{max}$. All holding times are sampled from the exponential distribution with rate λ_{max}. However, to make the uniformized chain stochastically identical to the original chain, we introduce transitions back to the same state. In the uniformized chain, the probability of making a transition from s to s' ($\neq$ s) is $q_{ss'}/\lambda_{max}$. The probability of making a transition back to s is $1 - \lambda(s)/\lambda_{max}$. Transitions of the latter form are known as *pseudo transitions*, as they do not affect the state of the Markov chain. The mathematical basis for uniformization is simply that a geometrically distributed sum (with mean $1/p$) of i.i.d. exponential random variables (with mean $1/\mu$) is itself an exponential, with mean $1/(p\mu)$. Whenever the original chain in state s; its holding time is exponential with rate $\lambda(s)$. Now suppose the uniformized chain (at rate λ_{max}) enters state s; the number of pseudo transitions that occur before actually leaving s is geometrically distributed with mean $\lambda_{max}/\lambda(s)$, and the distribution of time spent in s before leaving is that of a geometrically distributed sum of exponentials, each with mean $1/\lambda_{max}$. The effective distribution of time the uniformized chain spends in state s is exponential with mean $1/\lambda(s)$, just as in the original chain.

Let us now apply these ideas to a specific example. Consider a type of queue that has K servers, a Poisson source process with rate λ, and a service distribution that is a probabilistic mixture of exponentials: with probability p_f the service time is exponential with a fast rate μ_f, and with complimentary probability the service time is exponential with a slow rate $\mu_s \leq \mu_f$. Now imagine a queueing network with three such queues. We suppose that every departing job exits the system with probability p_d; conditioned on not departing the system, the job rejoins the same queue with probability p_r, and otherwise joins either of the other queues with equal probability. The state of one queue, say i, in this system can be described by a vector $s_i = (N_i, F_i, S_i)$, where N_i gives the total number of jobs in residence at the queue, F_i gives the total number of fast jobs in service, and S_i gives the total number of slow jobs in service. In the absence of any job transfers from other queues, the transition rate out of s_i is $\lambda_i(s_i) = \lambda + F_i\mu_f + S_i\mu_s$. The state of the entire system is the concatenation $s = (s_1, s_2, s_3)$, with total transition rate $\lambda(s) = \lambda_1(s_1) + \lambda_2(s_2) + \lambda_3(s_3)$.

Under an ordinary direct simulation of the Markov chain, the system remains in a given state s for an exponentially distributed period of time with rate $\lambda(s)$. After the holding time, the chain makes one of several possible transitions, chosen randomly. Transition due to a source arrival at queue i is chosen with probability $\lambda/\lambda(s)$, while transition due to a fast (alt., slow) service completion at queue i is chosen with probability $\mu_f F_i/\lambda(s)$ (alt., $\mu_s S_i/\lambda(s)$). Following simulation of the chosen transition and its effect on s, a new holding time is chosen based on the new state, and the simulation process continues.

An alternative form of direct simulation is more closely related to how we do parallel simulation. Let us now view the system as three interacting Markov chains, each one simulated directly. This is equivalent to partitioning all transitions into three classes, grouping together all transitions that are initiated at a common queue. We maintain a simulation clock t_i for each queue i, reflecting

the end of the queue's current holding time. To select the next event to do in the system we first select the queue i whose time t_i is least (recall the interpretation of a transition in terms of competing exponentials). We then directly simulate that queue, choosing a source arrival with probability $\lambda/\lambda_i(\mathbf{s}_i)$, a fast job departure with probability $F_i\mu_f/\lambda_i(\mathbf{s}_i)$, and a slow job departure with probability $S_i\mu_s/\lambda_i(\mathbf{s}_i)$. If a job departure is chosen, then with probability p_d the job leaves the system. If the job does not leave the system, then with probability p_r the job rejoins the same queue. Failing this, the job is routed to one of the other two queues, with equal probability. Queue i now has a new state $\mathbf{s}_i'$; its new next transition time is chosen by adding t_i to an exponential random variable with rate $\lambda_i(\mathbf{s}_i')$. Observe that if the event caused a job to be routed to queue $j \neq i$, then queue j has a new state $\mathbf{s}_j'$. We compute a new next transition time for queue j by adding t_i (**not** t_j!) to an exponential random variable with rate $\lambda_j/(\mathbf{s}_j')$. Also observe that if the event does not route a job to another queue, then the distribution of the holding times of the other queues are unaffected by the event, and, by the memoryless property of the exponential distribution, do not need to be resampled prior to selecting the next event. They *could* be resampled, but the resulting chain would be probabilistically identical to the one where we do not.

This description suggests that one might directly simulate each queue on a separate processor, provided one can accommodate the instances when jobs flow between queues. It is convenient to view the behavior of a given queue as the superposition of an *internal* stream of events, and a set of *external* streams. The internal stream is comprised of all events that do not directly affect the state of another queue: Poisson source arrivals, departures that leave the system, and departures that return immediately to the same queue. We have one (outgoing) external stream associated with each other queue; such a stream is comprised of all transitions that send a job to the associated queue. Now for each queue i we maintain a next internal transition time I_i, and two next outgoing external transition times E_{ij}, and E_{ik}, $j,k \neq i$. The queue's next transition time is the minimum, $t_i = \min\{I_i, E_{ij}, E_{ik}\}$. This is simply another application of the "competing processes" interpretation of a transition. When queue i is chosen for the next transition, then the event is taken from the stream whose next transition time is t_i. After simulating that event so that the queue enters state $\mathbf{s}'$, new next transition times are chosen for **all** streams associated with the queue. The reason for changing every stream's next transition time is apparent from the rates of these streams' holding times. The holding time rate for the internal process from state $\mathbf{s}_i = (N_i, F_i, S_i)$ is

$$\lambda_i^I(\mathbf{s}_i) = \lambda + (p_d + (1-p_d)p_r)(\mu_f F_i + \mu_s S_i), \tag{1}$$

while the holding time rate for either external process is

$$\lambda_{ij}^E(\mathbf{s}_i) = 0.5(1-p_d)(1-p_r)(\mu_f F_i + \mu_s S_i). \tag{2}$$

Equation 1 reflects arrivals to queue i (at rate λ), and service completions that either depart the system or are routed back to queue i. (Note that the total rate at which jobs in queue i are receiving service is $(\mu_f F_i + \mu_s S_i)$, a fraction p_d of which depart the system and a fraction $(1-p_d)p_r$ of which are routed back to queue i.) Both of these rates depend on the state of the queue; any event at that queue may change its state, and hence change the correct distribution for the next

event on each stream. New next transition times are computed by sampling from the new holding time distribution, and adding to the time of the event, t_i. If an external event is chosen, then new holding times must also be chosen in like fashion for all streams of the queue receiving the departing job.

Note that the transition rates above depend only on the number of fast and slow jobs in service. As such, these rates are independent of the queueing discipline, whose effect is manifested in the definition of the state transformation upon a job departure or arrival. A key point is that the transition rates are independent of the queueing discipline. This is important to remember, as it will imply that our synchronization algorithm is independent of the queueing discipline.

The problem remains that the instants when jobs leave one queue for another are erratic and unpredictable. We approach the problem by uniformizing every external event stream. Why? Because the holding time distribution of an external event stream then remains completely independent of any state changes that may occur at the queue. We can completely *presample* the holding times of all uniformized external event streams. Embedded in these transition times are real ones, where jobs move between queues. We do not know which of these transitions will actually move jobs and will not know until the simulation is actually performed, the queue states are actually known, and the real/pseudo decision thresholds are actually computable. The beauty of the method is that the queues can generate and exchange their external transition times, and then use these times as synchronization points, a.k.a. "appointments" [8]. Queue 1 presamples the *potential* transition times for its external event streams to queue 2 and queue 3. These potential times are sampled from Poisson processes whose rates, λ_{12} and λ_{13} are at least as large as the maximum possible instantaneous rates at which queue 1 can send jobs to queue 2 and queue 3. For example, $\lambda_{12} = 0.5(1 - p_d)(1 - p_r)K\mu_f$, which represents the rate at which jobs flow from queue 1 to queue 2 when all K servers on queue 1 are busy serving in the fast phase. By Equation 2, $\lambda_{12}^E(\mathbf{s}_i) \leq \lambda_{12}$ for all possible states $\mathbf{s}_i$. After queue 1 presamples these potential external transition times, it sends those lists to queue 2 and queue 3. Each queue i receives from every other queue j lists of times at which a job *might* be sent from j to i. These lists are merged with queue i's own lists of times when it may send jobs to other queues. The n^{th} entry in the merged list for queue i is of the form $(T_i(n), C_i(n))$, where $T_i(n)$ is the time of the n^{th} event and $C_i(n)$ is the type of the n^{th} event, i.e., $C_i(n) = (i, j)$ or (j, i), depending on whether the potential job goes from i to j or vice versa. Having done so, each queue now knows *each and every* time at which some other queue may affect it, and at which it may affect some other queue. Without uniformization the synchronization times are unpredictable; with uniformization they are completely pre-determined in advance of actually running the simulation.

As we simulate in parallel, each processor will execute asynchronously of the others, except for synchronization at the pre-arranged instants in time. For example, suppose that the state of queue i is $\mathbf{s}_i$, that the last event at queue i occurred at time t_i, and that $T_i(n)$ is the time of the next (potential) external event. An exponential holding time E_i with mean $1/\lambda_i^I(\mathbf{s}_i)$ is generated. If $t_i + E_i < T_i(n)$, then the next event to occur on queue i is an internal event. In this case, among all possible internal transitions, queue i chooses one with probability proportional to its

transition rate, simulates it, and updates its clock to time $t_i + E_i$. If $t_i + E_i \geq T_i(n)$, then the next event to occur at queue i is an external event. Suppose that $C_i(n) = (i, j)$. Then queue i decides whether the transition is pseudo or real by computing the ratio $r = \lambda_{ij}^E(\mathbf{s}_i)/\lambda_{ij}$ (the ratio of the stream's current actual rate to the stream's uniformized rate), opting for a real transition if a uniform $U(0, 1)$ random variable is less than or equal to r. In this case queue i selects a job whose service completes, selecting any particular job with probability proportional to the rate at which that job is departing for queue j ($0.5\mu_s$ or $0.5\mu_f$, depending on whether the job is fast or slow). Queue i sends a message to queue j specifying the job transfer and continues. If $C_i(n)$ is judged to be a pseudo (with probability $1 - r$), then queue i sends a message to queue j reporting this fact. Alternatively, if $C_i(n) = (j, i)$, then queue i waits for the message from queue j. If queue j reports a job arrival, then queue i simulates the arrival. If queue j reports a pseudo then the event does not affect queue i's state. Following simulation of $C_i(n)$, queue i advances its clock to time $T_i(n)$. A new holding time for the internal process is selected, and the process continues.

Observe that the description above serves to describe a general algorithm, if we merely replace the word "queue" with "LP". Also observe that it is possible to define windows $[t, t+\Delta]$ in simulation time. One generates and exchanges all uniformized external events that fall within the window, simulates the system behavior through that period, then advances to the next window $[t+\Delta, t+2\Delta]$. The only limitation on the window size Δ is the memory storage necessary to hold the external transition times.

Calculation of uniformization rates is always application dependent. Among all features of the algorithm, this is one of the issues demanding the most attention by the modeler to the synchronization algorithm. (The other major such issue is decomposing event streams into internal and external streams.) It is possible, for example, for LPs to be defined so that jobs from an infinite server queue are routed to different LPs. One can't bound the transition rate of such an external stream, at least not in an open system. The method works best when every stream's uniformization rate is very close to its actual job transfer rate, i.e., when most external events are real. However, this may not always be the case. Pseudo transitions are the single most deleterious artifact of the algorithm, because time spent generating, communicating, and synchronizing upon pseudos is time spent on activity not found in an optimized serial implementation. All of our PUCS variations were developed to reduce or eliminate sources of pseudo transitions, or to minimize their effect on performance.

Not all CTMCs are suitable for parallel simulation using PUCS. A key requirement is that one be able to partition the CTMC into loosely synchronous interacting subchains. Such partitioning follows intuitively when the CTMC has a basis in a physical domain, because partitioning the domain often has the desired effect. Nevertheless, the issue of defining suitable LPs automatically is one that we have not yet addressed.

The details above may seem complex, especially to those with little experience dealing with CTMC models. However, there is strong reason to believe that PUCS-style synchronization can be embedded in a parallel simulation package specific to an application class (e.g., a large subset of RESQ [3] for simulating queueing networks), with all the details of finding legal uniformization

rates being automated.

In the course of experimenting with PUCS we encountered several implementation issues. One of these concerns external stream list management. On the one hand we can faithfully implement PUCS as described above. On the other hand, we can avoid list transmission altogether, by having both ends of an external stream maintain a synchronized random number generator, so that LP i *computes* the time of the next LP $j \rightarrow$ LP i synchronization, rather than *receives* it. Another issue is the degree of aggregation one ought to employ when defining LPs. It is possible for the entire submodel assigned to a processor to be considered as a single LP. It is also possible to break up the model into more natural LPs, and treat the workload on a processor as a collection of distinct LPs. Yet another issue is whether to exploit optimism. The uniformization framework offers some unique optimizations for optimistic processing. Are they worth it? A final issue is that of adaptivity in uniformization rates. What can you do when a mathematically correct upper bound is either impossible or so large that almost all external transitions end up being pseudos? Our various implementations, to be described next, explore issues.

3 Methods

We describe five different methods based on uniformization, and give the rationale for each.

3.1 Conservative Aggregated PUCS

CA-PUCS (identified simply as PUCS in [5, 9, 10]) was one of the first methods we developed. In implementation it is almost identical to the description given in the last section. It has the additional characteristics that the entire submodel assigned to a processor is considered to be one LP, and that synchronization lists are generated and simulated on a window-by-window basis. The latter feature is needed for the simple reason that computers' memories can retain only a finite number of external transition descriptions, and very long runs will require very long transition lists.

The rationale for aggregating all co-assigned workload into one LP is two-fold. First, a one-LP-per-processor implementation is much easier to develop than one that allows multiple LPs. The architecture used in our studies—the Intel family of multiprocessors—supports interprocessor communication via explicit sends and receives. Receives may be either asynchronous (post a receive and periodically check on whether the anticipated message arrived yet) or synchronous (block until the anticipated message arrives). Furthermore, the Intel iPSC/860 and Touchstone Delta operating system, NX, supports only one process per processor. Any multitasking—like switching between LPs—has to be done at the application layer. By aggregating all of a processor's workload into one LP we avoid scheduling issues; furthermore, there is no need to buffer incoming communication at the application layer. When the processor expects message m at time t from processor j, it simply does a synchronous receive until that message materializes. One cannot use synchronous receives if switching between LPs is necessary. Secondly, massive aggregation avoids *internal* pseudo events

that may occur when multiple LPs are assigned to one processor. The problem here is that if uniformization is applied at the LP level, then two LPs on the same processor synchronize with each other just as though they were assigned to separate processors. We surely can develop the code so that the communication between co-resident LPs is cheap, but we cannot easily avoid the overhead of generating, communicating, and synchronizing upon a pseudo event. An important rationale for massive aggregation is to eliminate the possibility of internal uniformization.

3.2 Conservative Partitioned PUCS

The other side of the aggregation issue is that massive aggregation can cause artificial blocking. Events on a processor under CA-PUCS are executed in increasing monotonic order. If any piece of a processor's submodel needs a message at time t and if that message is not yet present, the entire processor blocks. However, it may be that another piece of the submodel is free to continue past time t. To block at time t is to cheat oneself of some potential parallelism.

CP-PUCS (identified as PUCSThreads in [9]) allows multiple LPs per processor, and also strives to reduce the communication overhead of list generation. The principle features of the method are

- **LP independence:** A processor may manage any number of distinct LPs. In addition, by appropriate assignment of random number generator seeds, the sample path that is executed can be made independent of the way in which LPs are assigned to processors.

- **Scheduling:** At any time, each LP is classified as being *ready* or *blocked*, depending on whether it is free to execute or is waiting for an incoming message. Scheduling consists of selecting the ready LP with least time-stamp, performing a communication (either a send or a receive) and simulating until it reaches its next communication instant. If an LP blocks waiting for a message, a description of that message is stored in a binary search tree. Between LP activations we probe for any newly received messages, accepting all such and storing them in the application space. As each new message is processed we examine the search tree to see if some LP is blocked on this message. If so, the LP is unblocked and placed on the list of ready LPs.

- **List Generation:** Every pair of LPs i and j maintain a synchronized random number generator. This means that LP i can compute for itself the same transition times that j computes for the LP j to LP i external stream. While each LP now executes more work by duplicating the generation of external stream transition times, we avoid having to communicate and merge the lists. There is an additional advantage in that no window is needed now to limit the memory usage of external transition times. We simply generate the "next" transition time for a stream when it is needed.

Somewhat to our surprise, our previous empirical studies found no real benefit of CP-PUCS over CA-PUCS. Those studies examined situations in which the deleterious effect of internal pseudos is the dominant bottleneck to achieving good performance, and thus the benefit of avoiding them outweighed the benefit of more parallelism. However, as we will see, data in the present paper

shows that this is not always the case and there are situations in which CP-PUCS outperforms CA-PUCS. We will comment more on this in Section 4.

3.3 Optimistic PUCS

Opt-PUCS (identified in [9] as OptAll) endows CP-PUCS with optimism. This comes into play when an LP reaches an incoming communication instant, and the message it is to receive is not yet present. The LP can optimistically assume that the message will report a pseudo transition, and hence there is no need to wait for it. When the message does finally arrive, if the receiving LP's guess was correct, then there is no need to roll back. This is an application of the idea of "lazy reevaluation" explored first in [13]. Otherwise, as with standard optimistic algorithms such as Time Warp[6], the receiving LP is rolled back to the time of the late message.

PUCS' general framework makes possible some unique optimizations.

- **State Certainty:** In a general purpose optimistic environment, one can never be certain whether the next event processed will end up being committed, or will be discarded as a result of rollback. In Opt-PUCS an LP can sometimes know that its state is **sure**, that it will not be rolled back past its present point. The key to this determination is that we know all instants in simulation time where messages may arrive. If LP i knows it will not receive any message between times s and t, and it knows that its present state is **sure** (all LPs are initially **sure**), then its state remains **sure** while processing all internal events up to time t. Furthermore, if LP j sends the message at t and was also **sure** at the time the message was sent, then the message may be received and LP i remains **sure**. However, if either LP j was **unsure** at time t, or if the LP i decides to optimistically bypass that communication, then LP i becomes **unsure**. In [9] we show how every LP can maintain a *Least Sure Time* (LST) that describes the last instant in simulation time when the LP was **sure**. By simply appending **sure/unsure** tags to messages and analyzing these, every LP's LST advances without extra calculation. Since we may release any state saved at a time less than the LST, the LST calculation gives us the benefits of the usual GVT (Global Virtual Time—see [6]) calculation, without the additional overhead of actually performing a GVT calculation.

- **State-Saving:** Optimistic simulations generally save state prior to every event, because as far as the LP knows, the simulation can in theory be rolled back to any point in simulation time ahead of the last known GVT. Within the PUCS framework, a rollback can occur only at some communication instant, hence there is no advantage to saving state before an internal event. The only time state must be saved is at a communication instant, and then only if the receiving LP is either **unsure** or becomes **unsure** by either receiving an **unsure** message or by optimistically bypassing it.

- **Scheduling:** Our ability to ascertain whether an LP's state is **sure** permits smarter scheduling than is usually possible under Time Warp because we may give highest priority to an LP with some work to do that we know is **sure**, and cannot be rolled back. In fact, our studies

in [9] found that a very effective scheduling strategy is one that is averse to state-saving, as follows. An LP's execution slice is delimited at either end by external communications (either incoming or outgoing); the execution slice begins by performing a communication, then all internal work up to (but not including) the next communication is performed. Whether or not we perform a state-save at the initial communication depends on the present **sure/unsure** state of the LP, whether the communication is outgoing or incoming, and whether an communication is present or **unsure**. We define four scheduling classes, listed below in decreasing order of priority.

1. **sure** LPs that will not save state because the first communication is either an incoming message from a **sure** LP, or is an outgoing message.
2. **unsure** LPs whose first communication is either an incoming message from a **sure** LP, or is an outgoing message.
3. **sure** LPs that must save state on the first communication, because that communication (necessarily incoming) is either not yet present, or was sent by an **unsure** LP.
4. **unsure** LPs that must save state on the first communication, because that communication (necessarily incoming) is either not yet present, or was sent by an **unsure** LP.

One of our aspirations for Opt-PUCS was that it would reduce the cost of pseudo transitions. While pseudos would still appear logically in the external event streams, the hope was that not having to communicate them from **unsure** LPs would lead to some savings. Our initial experiments showed that this intuition held true, provided that the fraction of pseudo events was very high. For lessor fractions of pseudos, the overheads of optimism largely cancelled the benefits of optimism. This observation is also borne out in the new data we present in this paper. One should also bear in mind that the version we study in this paper is highly optimized. Our previous study suggested that its performance is as large as a factor of 2 better than standard Time-Warp style algorithms.

3.4 Adaptive PUCS

We developed ACA-PUCS and ACP-PUCS in an effort to deal directly with the problem of excessive pseudo events. The idea is to *observe* the behavior of an external stream, and uniformize it at a rate slightly larger than the maximum rate it seems to achieve and repeat. There are two basic issues that must be addressed. One is the selection of uniformization rate, and the other is dealing with situations where the *assumed* upper bound on the external stream's transition rate actually becomes less than the *actual* transition rate—an occurrence we call a *rate fault.*

To uniformize a stream at a rate which is not provably an upper bound on its transition rate is to execute optimistically. Some provision must then be made to recover from faults suffered when optimistically made assumptions are violated. Our earlier experience with other versions of PUCS suggested that CA-PUCS was an appropriate point of departure, as it consistently achieved better performance (on the problems studied) when the fraction of pseudo events was low. The simplest way to incorporate optimism in CA-PUCS is to checkpoint the entire simulation state at

the beginning of every window, continuously monitor each external stream for rate faults, and at the end of a window determine w! ether any LP suffered a rate fault at any point in the window. If a rate fault is encountered the entire window is resimulated by all LPs. In order to correctly resimulate the window, exactly the same sequence of events must be performed up to the time of the earliest rate fault, say t. The uniformization rate of the faulting stream is increased just prior to time t to a level that will carry it passed the observed rate fault. The process of resimulating a window is repeated until we get through the window without any faults, at which point we advance to the next window. The net effect is that the uniformization rate for an external stream over a window is a piece-wise constant function of simulation time, with jumps occurring at instants when rate faults are observed on that stream.

We use a two-stage policy for determining a stream's initial uniformization rate, at the beginning of a window. During the first phase we monitor the stream's transition rate, and record the largest rate ever seen. While in the first phase, the initial uniformization rate given to the stream at the beginning of a window is twice the maximum rate seen so far. The second phase begins after the monitored maximum rate remains unchanged for a long time (more precisely, after the last consecutive 99% of the stream's transitions have past without a change). The stream then "locks in" on twice this maximum, uniformizing all subsequent windows at that rate. There is a provision to increase the lock-in rate, provided the fraction of windows that rate fault rises above 5%. The philosophy of this mechanism is to observe the stream's behavior for a long enough period of time so that the highest transition rate it is likely to see and return to is observed. We uniformize at twice this rate as a means of insurance. Once in the second phase, rare surges of the transition rate past the uniformized rate are accommodated via rate-faults and resimulation, but the default rate remains unaltered because the probability of exceeding that rate is very low.

Experiments with ACA-PUCS (reported in [10]) showed that it could indeed accommodate situations where non-adaptive PUCS failed. In the next section we present data that also shows this advantage. To complete our comparison of the influences that aggregation and adaptivity have on performance, we recently developed ACP-PUCS—an adaptive version of CP-PUCS. ACP-PUCS retains all the features of CP-PUCS; in addition, it handles adaptive uniformization in the same fashion as does ACA-PUCS. Windows are defined solely for the purpose of checking for and recovering from rate-faults. Prior to the adaptive mechanism locking on, the initiating end of a stream notifies the receiving end of the initial uniformization rate for the window. However, if that rate does not change between windows, then no such communication is needed. Consequently, once all the streams have locked in on their effective uniformization rate, the additional overhead associated with a window becomes negligible.

A final advantage of adaptivity is that it releases the simulation modeler from the burden of having to determine uniformization rates. For this reason adaptive methods seem to offer the most hope for automating the parallelization of a CTMC simulation.

4 Experiments

In this section we present the results of experiments performed on the Intel Touchstone Delta multiprocessor[7], using 16, 64 and 256 processors. The Delta is an MIMD architecture based on the Intel i860 processor chip. Its processors are connected in a mesh network. Communication is based on circuit switched message passing.

The simulation model we study is that of a fully connected network of central server queueing clusters [1]. A single central server is illustrated in Figure 1. A job entering the cluster always visits the CPU queue first. After receiving service there, the job is routed to one of twenty I/O servers, chosen uniformly at random. Upon entering service, the job chooses a "fast" service rate of μ_f with probability p_f; it otherwise acquires "slow" service rate of 1. The job receives an exponentially distributed amount of service, with mean $1/\mu_f$ or mean 1, depending on whether the job is fast or slow. Upon its service completion the job returns to the CPU server with probability p_c. Otherwise, some other central server cluster is chosen uniformly at random, and the job is routed to that cluster's CPU queue. Throughout our study of PUCS we have used this model, or another one related to it (where multiple local clusters are attached to each central server). Even though the model is too simple in and of itself to warrant treatment by parallel simulation, we use it because it is capable of parametrically representing more complex models. For example, the model parameter p_c can be used to adjust the computation/communication ratio. The performance of the synchronization protocol is largely independent of the specifics of the simulation workload. However, the frequency with which the model communicates and synchronizes obviously affects performance, and p_c is a simple parametric means of varying workload intensity. Similarly, the number of jobs circulating in the system is another parameter that affects the workload intensity. We can control the level of uniformization by adjusting μ_f—the higher it becomes, the faster the uniformization rates on external streams.

Our study sets $p_c = 0.99$. This implies a healthy computation/communication ratio proportional to 200 (an average of 100 visits to the CPU and some I/O device before exiting the cluster) -- but only in an "optimal" parallel simulation whose only communication costs are those of moving jobs. The actual ratio will be degraded from this level by uniformization. Because of the relatively high cost of message-passing, any application running on a machine such as the Delta must have a respectable computation/communication ratio to achieve respectable speedups. We also fix the probability of a fast job (p_f) to be 0.01. This selection places stress on our algorithms, because strict uniformization rates must assume that every server is always busy with a fast job, when in fact, fast jobs rarely appear. Our study fixes the number of central server clusters at 256. This selection gives us a moderately large simulation model, and also enables us to examine the effects of managing many LPs (up to 16) on a processor. Finally, we set the CPU service rate to 20, and the slow I/O job rate at 1. This ensures that in steady-state the distribution of jobs will be more or less uniform among all queues.

The parameters we vary are

- **Number of jobs:** We examine *lightly loaded* scenarios, where there are 10 jobs per cluster

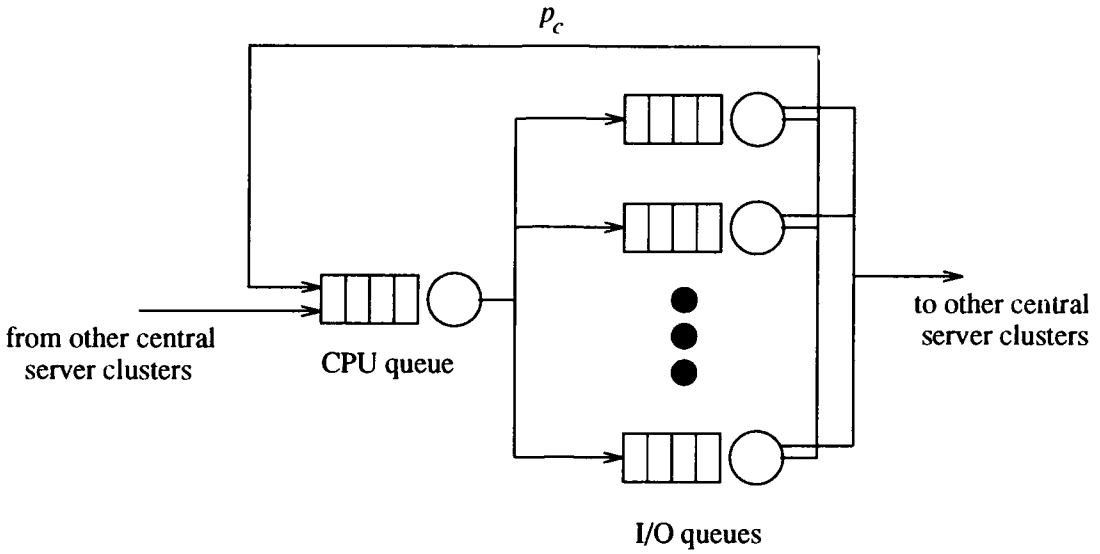

Figure 1: Central server model. p_c is the probability that a job departing an I/O device will return to the CPU queue.

(about 0.5 jobs/queue), and *heavily loaded* scenarios where there are 1000 jobs per cluster (about 50 jobs/queue).

- μ_f: We examine a fast job rate of 1 (so there is no distinction between fast and slow jobs), and a fast job rate of 8. The latter selection, coupled with with $p_f = 0.01$, induces high rates of uniformization relative to actual stream transition rates.

- **Number of processors:** We study our models on 4×4, 8×8, and 16×16 submeshes of the Delta.

Every experiment was run long enough so that every processor executes approximately 0.5 million events. Our primary metric of interest is the *event execution rate*, which measures the rate at which **useful** events are executed (per second). We specifically exclude from this rate pseudo events and optimistically executed events that are later rolled back. The rates we present are from single runs; this is justified, as in our experience there is very little variation (perhaps 1%) in these execution rates between runs of the same model.

While simple, the model we study presents a challenge to any performance oriented study, especially of a conservative synchronization algorithm. There is virtually no locality; a cluster is no more likely to communicate with a co-resident one than it is to communicate with an off-processor one. Every cluster communicates with every other cluster—there are approximately 2^{16} distinct communication paths to manage! Using P processors, every time a communication occurs there is a $(P-1)/P\%$ chance that the communication is between different processors. Furthermore, the model uniformization is consistent with a queueing policy where newly arriving fast jobs to preempt slow jobs. The *only* benign assumption made is that $p_c = 0.99$, an assumption needed to ensure

a sufficient computation/communication ratio. Finally, the maximal processor size, $P = 256$, is significantly larger than that used in most studies, and may be as large as any previous study using MIMD processors. The fact that we do achieve significant performance over optimized serial execution on a difficult problem proves the validity of our methods.

Before analyzing the results of our experiments, we address the issue of "speedup". Speedup is intended to measure the user's benefit of running the parallel algorithm. For this reason, one ought to compare parallel performance to that of an optimized serial algorithm. Some difficulties arise, however, when the serial *algorithm* which is optimal changes as the problem parameters of interest change. To illustrate the point, Table 1 below gives serial execution rates as a function of problem characteristics, for an optimized serial direct Markovian simulation, and CP-PUCS run on one processor. While PUCS on one processor is faster by almost 20% on one set of parameters,

(load,μ_f)	Optimized Serial Algorithm	PUCS on One Processor
(light,1)	6211	7014
(heavy,1)	6563	7706
(light,8)	6219	4166
(heavy,8)	6554	6469

Table 1: Execution rates (events/sec) of the optimized serial algorithm and PUCS running on one processor.

it is slower by 33% on another. By comparison, the optimized serial algorithm varies by only a few percent over these problems. A user is far more likely to choose a serial algorithm that is consistently good over one whose performance varies so widely.

Table 2 presents the results of our experiments. Without resorting to a definition of speedup, we *can* say that on the heavily loaded problem with $\mu_f = 1$ using 256 processors, CP-PUCS is 260 times faster than the particular serial simulator we used, and is 221 times faster than its own one processor implementation (and 14 times faster than its 16 processor implementation). In either case, it is clear that a very substantial improvement over serial execution is being achieved. As an additional point of comparison, we measured the execution rate of the commercial queueing network simulator RESQ [3], executing on an IBM 3090 mainframe. The model simulated by RESQ is actually substantially smaller than this one, having only 16 clusters. The RESQ execution rate is only 1,781 events/sec. Of course, one must take into account that RESQ is an industrial quality simulator able to handle a wide range of problems, whereas the PUCS code is handcrafted and optimized, with a much more restrictive domain. Nevertheless, this comparison illustrates parallel simulation's tremendous potential for accelerating solution times.

We next analyze this data with an eye towards addressing the issues of aggregation, communication costs, optimism, and adaptiveness.

	16 Processors		64 Processors		256 Processors	
	light	heavy	light	heavy	light	heavy
Fast Job Rate = 1						
CA-PUCS	80,032	102,504	301,765	411,362	985,327	1,575,146
CP-PUCS	109,585	122,186	378,329	393,418	1,043,609	1,709,567
Opt-PUCS	103,707	121,609	343,510	353,873	874,617	855,737
ACA-PUCS	79,711	102,329	311,168	403,380	989,351	1,567,038
ACP-PUCS	78,942	100,877	256,138	327,559	808,621	1,357,975
Fast Job Rate = 8						
CA-PUCS	53,339	76,580	181,785	299,660	668,323	1,147,282
CP-PUCS	58,753	90,708	202,205	311,920	457,252	934,120
Opt-PUCS	57,314	89,642	167,382	328,711	445,880	802,732
ACA-PUCS	74,580	90,857	258,018	352,763	851,204	1,304,502
ACP-PUCS	63,738	88,156	203,168	282,835	547,770	926,403

Table 2: Execution rates (events/sec) of fully connected model of 256 central server clusters with $p_c = 0.99$, $p_f = 0.01$. Fast job service rate is varied between 1 and 8; average number number of jobs per cluster is varied from 10 (light) to 1000 (heavy). Simulation is executed on 16, 64, and 256 processors of the Intel Touchstone Delta.

4.1 CP-PUCS vs CA-PUCS

Our earlier studies of CA-PUCS and CP-PUCS (on an Intel iPSC/2) indicated that the CP-PUCS overheads of managing multiple LPs and of internal pseudos between on-processor clusters outweighed the advantages of increased opportunity for parallelism and avoidance of synchronous appointment generation. Yet the data in the present study shows that this is not always true. Consider Table 3 which gives the ratio of CP-PUCS rates to CA-PUCS rates, as a function of problem characteristics and architecture size.

The overall trend is for CP-PUCS to outperform CA-PUCS, but there are still instances where the reverse is true.

(load,μ_f)	16 Processors	64 Processors	256 Processors
(light,1)	1.37	1.25	1.05
(heavy,1)	1.19	0.95	1.08
(light,8)	1.10	1.11	0.68
(heavy,8)	1.18	1.35	0.81

Table 3: Ratio of CP-PUCS/CA-PUCS execution rates.

CP-PUCS and CA-PUCS differ both with respect to aggregation, and with respect to message handling. As such, it is difficult to separate the influences of aggregation and communication costs. Furthermore, the communication costs will depend on the underlying architecture, as well as the operating system. There are at least four factors to take into consideration, which sometimes interact in a complex manner.

- An LP's execution time-slice is delimited by communication instants. When $\mu_f = 8$ the uniformization rate is eight times larger, so that there are eight times as many communication instants per unit time. An LP's execution time-slice is much shorter, so that the overhead of switching between LPs is suffered eight times as often.

- In the lightly loaded experiments (and those where $\mu_f = 8$), most communications report pseudo events. Thus, when CA-PUCS blocks, it usually waits for a communication that doesn't affect its state. There is thus no useful purpose gained by blocking, other than the assurance of logical correctness. CP-PUCS is better able to find and execute useful work, when such work exists.

- As we increase the number of processors we decrease the number of clusters on a processor. This increasingly limits CP-PUCS' ability to find useful work that CA-PUCS cannot find. Of course, at 256 processors, both CP-PUCS and CA-PUCS each have one cluster per processor, and thus behave identically with respect to synchronization.

- CA-PUCS has a global step where synchronization appointments are generated and exchanged. Its performance will thus be affected by the efficiency with which an all-to-all exchange can be performed, and by the frequency of this exchange. CP-PUCS has no corresponding cost.

Let us examine performance with these factors in mind. On these experiments CP-PUCS tends to perform better. Apparently, on this model, the scheduling and appointment generation advantages outweigh CA-PUCS advantages. The difference between the two tends to diminish as the number of processors increases, which is consistent with the fact that (i) the CP-PUCS scheduling advantage gets smaller as a processor has fewer and fewer clusters, and (ii) in a CA-PUCS appointments exchange, essentially the same communication workload is spread over more network hardware, reducing the frequency of collisions and blocking. Thus, as the number of processors increases the CA-PUCS advantage diminishes and the CP-PUCS disadvantage diminishes. However, there are clearly other factors at work, as the performance differences change neither smoothly nor monotonically as the number of processors increase.

Our earlier comparison of CP-PUCS and CA-PUCS found CA-PUCS to be clearly superior. One explanation is that the models studied are different in an important way. The earlier model appends 10 "local clusters" of queues to every central server queue. In those studies, $p_c = 0.0$, and a job leaving an I/O device can be routed either to another central server cluster (with probability p_{cc}) or to one of its local clusters. Upon leaving the local cluster the job returns to the same

(load,μ_f)	16 Processors	64 Processors	256 Processors
(light,1)	1.05	1.10	1.19
(heavy,1)	1.00	1.10	1.99
(light,8)	1.02	1.20	1.02
(heavy,8)	1.01	0.90	1.16

Table 4: Ratio of CP-PUCS/Opt-PUCS execution rates.

central server. This model provides another way of boosting the computation/communication ratio, because a local cluster is always mapped to the same processor as its parent central server cluster. Our previous study varied the probability p_{cc} of routing a job from one central server to another one, on a different processor. As p_{cc} increases, CP-PUCS performance drops faster than that of CA-PUCS, because CP-PUCS suffers increasingly from internal pseudo transitions between a central server and its local clusters. The present set of experiments are somewhat kinder to CP-PUCS, as the level of interaction between co-resident LPs is much lower. It seems then that the level of internal uniformization is the deciding factor between CA-PUCS and CP-PUCS. This implies that close attention must be paid when partitioning a simulation model into LPs for PUCS, perhaps deciding which style of synchronization to use as a function of uniformization rates.

4.2 Whither Optimism?

These experiments offer clear insight into the potential of exploiting optimism in PUCS, because the only substantive difference between CP-PUCS and Opt-PUCS is the optimistic processing. Towards this end, Table 4 computes the ratio of CP-PUCS to Opt-PUCS execution rates.

The first thing we notice is that CP-PUCS tends to do a little better than Opt-PUCS. Next we notice is that the degree to which CP-PUCS does better tends to increase as the number of processors increases. Indeed, for all practical purposes the performance on 16 processors is identical; yet at 256 processors, in one case CP-PUCS was nearly twice as fast as Opt-PUCS.

Explanations for this behavior are found by looking at the costs suffered by executing optimistically, primarily event re-execution and state-saving. Table 5 computes the ratio of the number of total events (excluding pseudos) executed to the number of events (excluding pseudos) committed. One can also view this as the average number of times a non-pseudo event is executed. The table also computes the average number of state-saves per committed non-pseudo event.

One thing clearly shown is that, in this example, the cost of saving the state of one central server cluster (about 3000 bytes) is usually amortized over many events. Its effect on performance must be negligible. Any significant differences between CP-PUCS and Opt-PUCS are related to the cost of rolling back and re-executing events. Indeed, there is a direct correlation between high event execution ratios and significant gaps between CP-PUCS and Opt-PUCS.

Since re-execution costs define the difference between CP-PUCS and Opt-PUCS, it is simple

	Total/Committed Events			Average State Saves/Event		
(load,μ_f)	16 Processors	64 Processors	256 Processors	16 Processors	64 Processors	256 Processors
(light,1)	1.11	1.19	1.68	0.008	0.010	0.027
(heavy,1)	1.03	1.40	2.10	0.001	0.002	0.007
(light,8)	1.01	1.06	1.34	0.060	0.100	0.017
(heavy,8)	1.01	1.04	1.27	0.004	0.015	0.041

Table 5: Overheads associated with Opt-PUCS.

to explain why the gap between them increases as the number of processors increases. On only 16 processors, many LPs are assigned are assigned to the same processor, and thus Opt-PUCS has a good chance of being able to schedule a **sure** cluster. However, for a large number of processors there are relatively few LPs on a processor. Without a large number of LPs, a processor quickly executes its **sure** workload and is left to forge ahead optimistically. Apparently its optimism is frequently misplaced, and significant fractions of events end up being resimulated. This effect is somewhat lessened when there are many pseudo events, since in such cases the optimistic assumption that the event is a pseudo event is in fact correct.

4.3 Adaptivity

Pseudo-events are the largest source of performance degradation in all versions of PUCS. Many CTMC models have characteristics that cause the best upper bound on an external event stream's transition rate to be very far from the stream's average transition rate. In our experiments fast jobs appear infrequently, and one almost never sees more than 3 simultaneous fast jobs in a central server cluster. Yet the uniformization bound must be based on the assumption that all servers are busy with fast jobs.

Table 6 illustrates the sensitivity of each method to increased uniformization, by computing the ratio of its execution rate using $\mu_f = 1$ to its rate using $\mu_f = 8$. This data shows clearly that ACA-PUCS and ACP-PUCS are more tolerant of increased uniformization than are the other methods (with the exception of ACP-PUCS using 256 processors). Similar observations held in our previous study of ACA-PUCS that varied μ_f more widely, up to $\mu_f = 1024$. Even at levels of $\mu_f = 256$, ACA-PUCS gives respectable performance while CA-PUCS performance has thoroughly degenerated. We believe that any standardized version of PUCS must include adaptivity if it is to work on a wide range of problems.

The relatively weak performance of ACP-PUCS surprised us, as we expected it to gain the advantages of both scheduling flexibility, and adaptivity. We have reason to believe that its failure to do so rests somehow with the Delta architecture and NX operating system, because these expectations *are* meet using the Intel iPSC/2. Execution rates taken from a 16 processor configuration

Algorithm	Light Load			Heavy Load		
	16 Processors	64 Processors	256 Processors	16 Processors	64 Processors	256 Processors
CA-PUCS	1.50	1.66	1.47	1.34	1.37	1.37
CP-PUCS	1.86	1.87	2.28	1.34	1.25	1.83
Opt-PUCS	1.80	2.05	1.96	1.35	1.07	1.06
ACA-PUCS	1.07	1.20	1.16	1.12	1.14	1.20
ACP-PUCS	1.23	1.14	1.26	1.16	1.47	1.46

Table 6: Ratio of $\mu_f = 1$ to $\mu_f = 8$ execution rates.

	light	heavy	light	heavy
	Fast Job Rate = 1		Fast Job Rate = 8	
CA-PUCS	10,637	13,431	7,788	11,216
CP-PUCS	13,149	15,329	9,258	13,224
ACA-PUCS	10,679	13,212	8,118	10,553
ACP-PUCS	12,975	15,148	11,276	13,935

Table 7: Execution rates on 16 processors of Intel iPSC/2

are given in Table 7. We see that when $\mu_f = 1$ ACP-PUCS gets very nearly the performance of CP-PUCS (whose performance is best), while ACA-PUCS does not do as well owing to its basis in CA-PUCS. Then, when $\mu_f = 8$, ACP-PUCS becomes the best method over all.

Regardless of whether ACP-PUCS meets our expectations or not, it *is* evident that adaptiveness offers performance gains for $\mu_f = 8$, when the gap between the maximum and average external transition rates increases.

5 Conclusions

This paper looked at the problem of parallelizing the simulation of continuous time Markov chains. We showed how the notion of uniformization can be applied so that the simulation can be conducted by essentially pre-computing an inter-LP synchronization schedule, and then simulating a mathematically correct sample path through that schedule. This basic method is called PUCS. We described five different PUCS variations, and examine performance on a parameterized model designed to illustrate their respective strengths and weaknesses. The experiments were conducted on the Intel Touchstone Delta multiprocessor, using 16, 64 and 256 processors.

The results of these experiments, taken in conjunction with others previously conducted, suggest that an optimized PUCS algorithm ought to incorporate conservative synchronization, and

adaptive uniformization rates. Issues of aggregation and communication seem to be dependent on the simulation model, and underlying architecture and/or operating system. More work is needed to fully understand the complex relationships between these factors. The performance we observe can often be quite good, depending on the problem characteristics. However, PUCS performance *is* inescapably dependent on the number of pseudo-events, and every effort must be made to reduce these.

While our experiments prove the promise of PUCS, some important issues remain open. We have not yet addressed automated partitioning, nor automated load balancing, nor the effect one has on the other. We intend to investigate these issues.

Acknowledgements

This research was performed in part using the Intel Touchstone Delta System operated by Cal. Tech. on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the NASA Langley Research Center.

References

[1] J.P. Buzen, "Computational Algorithms for Closed Queueing Networks with Exponential Servers," *Commun. ACM*, vol. 16, no. 9, pp. 527-531, September 1973.

[2] R.M. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, vol. 33, no. 10, pp. 31-53, 1990.

[3] K.J. Gordon, R.F. Gordon, J.F. Kurose and E.A. MacNair. "An Extensible Visual Environment for Construction and Analysis of Hierarchically-Structured Models of Resource Contention Systems," *Management Science*, vol. 37, no. 6, pp. 714-732, June 1991.

[4] D. Gross and D.R. Miller, "The Randomization Technique as a Modeling Tool and Solution Procedure for Transient Markov Processes," *Operations Research*, vol. 32, no. 2, pp. 343-361, March-April 1984.

[5] P. Heidelberger and D.M. Nicol, "Conservative Parallel Simulation of Continuous Time Markov Chains Using Uniformization. IBM Research Report RC16780, Yorktown Heights, New York, 1991. To appear in *IEEE Transactions on Parallel and Distributed Systems.*

[6] D. R. Jefferson, "Virtual Time," *ACM Trans. on Programming Languages and Systems*, vol. 7, no. 3, pp. 404 - 425, July 1985.

[7] S.L. Lillevik, "The Touchstone 30 Gigaflop DELTA Prototype", *Proceedings of the 1991 Distributed Memory Computer Conference*, IEEE Press, pp. 671-677, April 1991.

[8] D.M. Nicol, "Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks," *Proceedings of the ACM/SIGPLAN PPEALS 1988. Parallel Programming: Experiences with Applications, Languages and Systems.* ACM Press, pp. 124-137, 1988.

[9] D.M. Nicol and P. Heidelberger, "Optimistic Parallel Simulation of Continuous Time Markov Chains Using Uniformization", IBM Research Report RC17932, Yorktown Heights, New York, 1992. Submitted for publication.

[10] D.M. Nicol and P. Heidelberger, "Parallel Simulation of Markovian Queueing Networks Using Adaptive Uniformization", IBM Research Report RC18403, Yorktown Heights, New York, 1992. Submitted for publication.

[11] R. Righter and J.V. Walrand, "Distributed Simulation of Discrete Event Systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 99-113, January 1989.

[12] S. Ross, "Stochastic Processes", John Wiley and Sons, New York, 1983.

[13] D. West, Lazy Rollback and Lazy Reevaluation, M.S. Thesis, University of Calgary, January 1988.

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1992	Contractor Report

4. TITLE AND SUBTITLE
PARALLEL ALGORITHMS FOR SIMULATING CONTINUOUS TIME MARKOV CHAINS

5. FUNDING NUMBERS
C NAS1-18605
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
David M. Nicol
Philip Heidelberger

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 92-60

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-189729
ICASE Report No. 92-60

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report

Submitted to the 7th Annual Workshop on Parallel and Distributed Simulation

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

We have previously shown that the mathematical technique of uniformization can serve as the basis of synchronization for the parallel simulation of continuous-time Markov chains. This paper reviews the basic method and compares five different methods based on uniformization, evaluating their strengths and weaknesses as a function of problem characteristics. The methods vary in their use of optimism, logical aggregation, communication management, and adaptivity. Performance evaluation is conducted on the Intel Touchstone Delta multiprocessor, using up to 256 processors.

14. SUBJECT TERMS
Markov chains; parallel simulation; parallel algorithms; queueing networks

15. NUMBER OF PAGES
23

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

NASA Langley 1992